CS 249 Project 3
Recursion

Overview

This project requires the completion of the following tasks:

- Create an implementation of Merge Sort to recursively sort items
- Create a program to solve the N-Queens problem using recursion
- Create a program to solve the 1/0 Knapsack problem using recursion

Unit Testing

This project requires that for each part you must run the included unit tests to validate and verify the correctness and integrity of your programs, as well as to drive a test-driven development approach. Remember that test-driven development means that test and compilation errors are not necessarily a bad thing; you should use them as an indication of you what code you need to create or fix next.

Start by downloading the interface and unit tests files provided for this assignment and setting-up JUnit for your specific Java development setup. Below are some quick tips for getting started running JUnit tests in several common development setups.

After you turn in your assignment you will be graded using these tests, as well as additional unit tests, so make sure you are thorough in your implementation.

IntelliJ:

- Create a new project and add the interface and test files to the *src* directory
- Open up one of the test files from within IntelliJ and click on one of the lines underlined with an error
- Press ***alt+enter*** (***option+enter*** on mac) and select the option to import the JUnit library into your project
- You should now be able to run each of the tests (***right-click*** the file and choose *run*), or all of the tests (***right-click*** on the project and choose to run all tests) once you create the classes required for this project

Eclipse

- Create a new project and add the interface and test files to the project *src* directory (you might need to ***right-click*** and refresh the *src* icon in Eclipse)
- ***Right-click*** on the project in the Package Explorer panel, go to ***Build Path->Add Libraries***, and select JUnit
- You should now be able to run the unit tests by ***right-clicking*** on the project or test files under Project Explorer, or by going to the ***run->run as*** menu

Command Line

For a command line development environment, you will need to perform several steps:

- Download the required JUnit JAR files to your project directory
- Add the JAR files to your project class path when compiling and running
- Create your own driver class (a class with a main method) to run the tests and print out the results

Merge Sort

Merge sort is an algorithm which recursively divides the array into smaller and smaller arrays, and then merges those arrays into each other until the entire set is merged into a sorted order. Remember that as you recursively divide the array you don't need to move the array in memory. You can find a complete description of this algorithm in your textbook as well as online.

1. ***You must write a MergeSorter class implementing the interface contained in IMergeSorter.java***
    - Remember to set up Merge Sort's base case. A length 1 or 0 array does not need to be sorted, and returns 0 operations.
    - The returned operation count for this sorter is defined as the number of elements that must be merged together during the merge step in each process, summed over all levels of recursion. For example, if you must sort the list [3, 2, 1, 5, 4] you must first sort the lists [3,2] and [1, 5, 4], which also requires you to sort the lists [1] and [5, 4]. Merging [5], [4] into [4,5] takes 2 operations, merging [1], [4, 5] into [1, 4, 5] takes 3 operations. Merging [3],[2] into [2, 3] takes 2 operations. Merging [2, 3], [1, 4, 5] into [1, 2, 3, 4, 5] takes 5 operations. All together this process takes 12 operations.
2. ***Create a main method which allows you run the MergeSorter for randomly generated numeric arrays of lengths 20,000, 100,000, 500,000, and 2500,000. Record the run times for each call.***
    - Use the java's System.nanoTime() to record the timing for each call.
    - Include a graph of these run times and an analysis of the growth rate in your write-up.

The N-Queens Problem

In chess the Queen is valued as the most powerful piece on the board, capable of moving any number of squares vertically, horizontally, or diagonally. The N-Queens problem proposes trying to place N Queens on an NxN board without any of the queens threatening one another. For a standard chess board N is equal to 8, where you must be able to find a place for 8 queens such that none of them are threatening one another, such as the situation shown in Figure 1.

In this project you are to implement a recursive brute-force solution to this problem, checking every possibility until you find a solution. Place each queen in the first available location and backtrack only when you discover that a particular queen placement is impossible to find a solution for.

This is an example of a depth first search with backtracking, as you explore, in full, the possibilities of each choice in sequence.

3. ***You must write an NQueensSolver class implementing the interface contained in INQueenSolver.java.***
    - The interface contains two methods to be complete, on which acts as an entry point, and another which is designed to be called recursively. Normally the recursive call would be private, but we will expose it for testing.
4. ***Create a main method which allows you run the NQueensSolver for N values from 4 to 12. Run each value of N 100 times, and record the run times for each set of calls.***
    - Use the java's System.nanoTime() to record the timing for each call.
    - Include a graph of these run times and an analysis of the growth rate in your write-up.

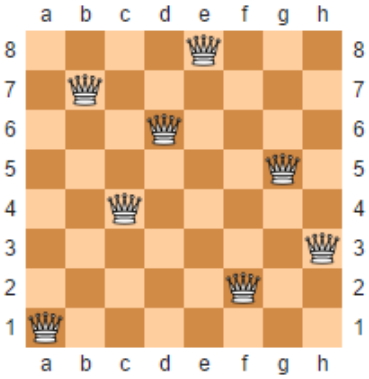

Figure 1 One solution to the 8 Queens problem

The 1/0 Knapsack Problem

The Knapsack problem asks you to pick from a set of items to find the combination with the highest total value under some weight limit. Assume you have a list of items, each of these items with some weight and some value. You also have a knapsack, or backpack, with a specific weight limit. You must determine which items you should take to not exceed the weight limit, while maximizing the value contained in the knapsack.

The recursive solution is to pick an object and then to perform the knapsack problem again with the remaining items and the remaining storage capacity. Once your storage capacity is overloaded you unroll and select the items which lead to the greatest total value. Like the N-Queens problem, this is an example of a depth first search with backtracking.

5. ***You must write a Knapsack class implementing the interface contained in IKnapsack.java.***
    - An inner class is defined for you called Item inside of the IKnapsack interface. This models an item that could be placed in the knapsack. An array of these will be passed to your solution method, and you must pick which items to keep, and which to leave.
    - You need only return the highest value solution. Should there be multiple even solutions, either is valid.
6. ***Create a main method which allows you run the Knapsack problem for random items of quantity 10, 15, 20, and 25. Record the run times for each call.***
    - Each randomly generated item should have a weight and value between 0 and 9. The capacity for each run should be 3 times the number of items.
    - Use the java's System.nanoTime() to record the timing for each call.
    - Include a graph of these run times and an analysis of the growth rate in your write-up.

Write-up and Submission

- For each part of the project, describe your approach, solutions, describe any difficulties that you encountered, and detail the efficiency of all non-test methods in your code using Big-O notation. Include any requested output and screenshots for each part.
- Add all of your *.java* files and your write-up document to a .zip or .tar.gz archive and submit it on Bb Learn. Make sure all of your java files are in the root of the zip file, rather than inside additional folders.
- You **must** write and submit your own code. You must also clearly identify and online or text sources that you used as references, any collaborators with which you discussed the project (or lack thereof), and how the source was beneficial.

Grading

This project is worth 100 points.

- 10 Points – Design and code quality
  - Good object-oriented design, consistent comments, white space, indentation, etc.
- 60 Points – Implementation quality
  - Demonstrate that your solution is complete and accurate based on unit tests.
  - 20 points – Merge Sort
  - 20 points – The N-Queens Problem
  - 20 points – The 1/0 Knapsack Problem
- 30 Points – Write-up and Submission
  - 5 points – Merge sort Analysis
  - 5 points – N-Queens Analysis
  - 5 points – Knapsack Analysis
  - 15 points – Other